UNIT 5. INTENSIVE RECURSION (BACKTRACKING)

1. Introduction.

Many problems can be formulated in terms of graphs, understanding by graph, a structure of data in the memory of a computer, representing nodes and edges in said memory and on which a series of quite specific operations are carried out. In these cases, if algorithmic methods are available to solve the problem (as we have seen in previous units), these algorithms require the visit of all the nodes and/or all the edges and a certain order in the routes.

Other times this is not the case. In other cases, the graph only exists implicitly, due to its large size. This is the case of graphs of games such as chess. The nodes would be the possible situations and the edges the valid plays. Most of the time, all we have is a representation of the current situation (the current node we visited) and a small number of different situations. In any case, the techniques used to cover them are the same.

An implicit graph is a graph for which a description of its nodes and edges is available so that relevant parts of the graph can be built as the path progresses. The time saving is great if the route is successful before having built the entire graph. The memory savings are also great if we can discard the nodes that have already been examined. For this we will use the technique of backtracking or intensive recursion.

2. Exhaustive search.

If a problem does not have an algorithmic method that solves it, an exhaustive search of solutions will be carried out among all the possibilities. This method is known as brute force: all possible cases are generated and tested one by one until finding the necessary solutions (sometimes it is enough to find one, other times you have to find all, and sometimes the best).

However, many of these problems can be solved in stages. Step by step, you check if a solution is being created or if decisions have been made that will not solve the problem. At each stage the properties whose validity is to be examined are studied in order to select the appropriate ones to proceed to the next step. The management of the stages, the possible decisions that are made and the relations between them, assume the generation of a decision tree. The partial solutions are in the nodes, and the links between them are the decisions taken to change the stage. The progress along the tree stops when it comes to a situation in which no decision can be made to obtain a solution, or when the problem is actually solved. In these cases, a previous partial solution is recovered and the search is continued if necessary.

3. Backtracking.

In its basic form, the return is a similar route to a search in depth in a directed and acyclic graph. It is usually a tree (implicit). The normal thing is to use the preordered route. The objective of this journey is to find solutions for some problem and is achieved by building partial solutions as the journey progresses; this is limiting the regions in which a complete solution can be found. The route is successful if, proceeding in this way, a solution can be completely defined. In this case, the algorithm will stop (if we only look for a solution)

or it will continue looking for alternative solutions (if we look for all of them). The route is not successful if at some stage the partial solution that we are building cannot be completed. In this case, the route goes back in depth, eliminating on the fly the elements that were added in each phase. When you return to a node that has one or more unexplored neighbors, the route of a solution continues.

In other words, backtracking is a recursive method to search for solutions in stages exhaustively, using trial and error to obtain the complete solution of the problem by adding decisions to the partial solutions. For this, it uses the organization of the recursion as a decision tree. To move forward a stage, a recursive call is made, and when to go back, what is done is to end the corresponding recursive process, which effectively returns to the previous state due to the stack of environments created in the recursion. In that case, the decision tree is not a structure stored in memory, but an implicit tree that is usually called an expansion tree or search space.

In its basic form, the expansion tree is pre-ordered: first the current or root node is evaluated, and then all its children from left to right. Therefore, until a partial solution is generated (valid or not), no other solution is evaluated. In the nodes of the level k of the tree are the partial solutions formed by k stages or decisions. We must remember that the tree is not really stored in memory, it is only traversed at the same time it is generated, so everything that wants to be preserved (decisions taken, solutions already found to the problem, etc.) must be stored in some additional structure. Finally, because the number of decisions to be taken and evaluated can be very high, if possible, it is convenient to have a function that determines if a complete solution can be reached from a node, so that using this function it is possible to avoid going through some nodes and, therefore, reduce the execution time.

The basic scheme of the Backtracking approach is as follows (k represents the level, and sol the current partial solution, the initial call to the Backtracking procedure would receive as parameters an empty solution and value of k equal to 1):

```
proc Backtracking(E/S sol: vector; E k: natural)          {1}
    var sons: list of decisions
    sons = créate_level(k)                                {2}
    for i = 1 to total(sons) do
        sol[k] = sons(i)                                  {3}
        if is_solution(sol,k) then
            treat_solution(sol)                           {3a}
        else
            if is_completable(sol,k) then
                Backtracking(sol,k+1)                     {3b}
            eif
        eif
    efor
eproc
```

This scheme corresponds to the following ideas:

1. When arriving at a node (when entering a recursive call) you have a partial solution formed by k-1 stages.
2. All possible decisions are generated from the current partial solution.
3. For each of these decisions, it is incorporated into the partial solution (so now it will have k stages) and it is checked if a total solution has been achieved to.
    a. If it is a solution to the problem, it is managed: it is stored in a solution structure, compared with previous ones, it is shown to the user ... and the search ends with those decisions.
    b. If it is not a solution, if it is detected that the partial solution is complete (in case of having a function capable of detecting this situation) a new recursive call is made with the new partial solution and the new level.

Many components can be incorporated into this basic scheme, generally in the form of parameters that are added in the recursive call:

- If all we were looking for was a solution to the problem, the algorithm should end at point (3a). For this, an output boolean parameter can be used that is marked TRUE at this point, and that is used with a conditional in (3) to see if it is necessary to continue adding decisions or not.
- If all the solutions were wanted, the recursive procedure will need another input/output parameter that will be the structure where all the found solutions are stored, for example, a list that is updated in (3a).
- If the best solution to the problem was sought (as in optimization problems), the procedure will need another input/output parameter that stores the best known solution so far. When finding a new solution in (3a) it is compared with the one that is passed by parameter, updating it if necessary, and continuing with the search.
- If the creation of the children in point (2) is expensive, and the set of decisions that can be taken in each stage k is always the same, a new parameter is added to the method to keep track of the decisions that they can be taken at any time, for example, by means of a set or vector. When a decision is made in section (3), it is eliminated from the set so that the children cannot take it again.

Costs and efficiency in Backtracking: The way to generate and choose between the different possibilities determines the shape of the tree, the number of descendants of a node, the depth of the tree, etc., and determines the number of nodes of the tree and possibly the efficiency of the algorithm, since the execution time depends to a large extent on the number of nodes that are generated and visited. Usually, there will be at most as many levels as there are values in a solution sequence. Assuming that there are n stages and m possible decisions in each of them (normally m is greater than n) the order of efficiency of Backtracking is in $O(m^n)$; even when you can have m=n you get the potential order $O(n^n)$. In general, the efficiency depends on:

- the time necessary to generate the possible decisions in point (2),
- the number of possible decisions that are obtained in (2),
- incorporate a decision to a partial solution in point (3),
- the number of partial solutions that satisfy is_completable,

- and to a much lesser extent, the time it takes to check is_solution and treat_solution.

The costs can be improved if create_level has some boundary function to generate solutions (for example, not based only on the current solution but in several steps "ahead") that reduce the number of nodes generated. However, it must be borne in mind that a good dimension predicate can require a lot of evaluation time. If it is reduced to a single generated node, the Backtracking method becomes a voracious solution: there is a total in O(n) nodes to be generated in total. If each node has all the possible decisions, but so that they cannot be repeated at different levels, the cost approaches O(n!). If possible, try to reorder the selections so that the number of C_i decisions that can be taken at level i is increasing, that is, $C_1<C_2<...<C_n$. The idea is that in the first cases there are less possibilities to evaluate so that the tree does not expand too soon.

The main problem with this technique is that having a large recursion load and due to the high step of parameters between the calls, a series of considerations must be taken into account. In the first place, passing parameters only as input data consumes memory and time, since local copies are made for each of the environments. It can be avoided with the use of parameters by reference, but we must be careful with the modifications since they will affect at all levels. As the search space is implicit, it may be necessary to redo nodes that have already been resolved, and which will have to be recalculated. One way to avoid it (although it depends on the problem to solve) is to generate decisions sequentially and avoid reordering (in certain problems, take decision 1 and then decision 2 can be equivalent to taking decision 2 first and then decision 1) . If possible, a parameter is often incorporated into the recursive call to indicate the range of valid decisions.

Because recursion is a form of programming that generates independent environments (different states in the recursion stack), the most serious error when using Backtracking is corruption. Corruption is understood as the fact that modifications made in one environment involuntarily affect another. There are two types: corruption between siblings and corruption from parents to children. Sibling corruption occurs when a parent deals with their child nodes differently (point 3 of the schema). The information provided by the father must be the same for all children; the only thing that must change is the incorporation of the corresponding decision to each of them, and if this is not the case, the creation of environments is incorrect. There are two ways to avoid this problem:

- Use auxiliary variables that are a copy of the father's environment and pass them to the children. Thus the environment of the father is “safe” since the modifications are made on the auxiliary environment, and for the following child the original conditions are preserved.
- If, when creating a new child, the environment of the father is modified to pass it to the recursive call, when returning from it, the actions that have produced changes for the next child must be undone, and the decisions of his previous brother should not be affected.

Corruption from parents to children (which actually occurs in the children and affects their father) is due to the passing of parameters by reference: as the parameter is the same within the whole decision tree, the changes that are made in the children are transmitted “upwards” when the recursion ends, thus affecting the father. There is no general way to avoid this problem except caution.

4. The problema of the eight queens.

On a chessboard (size 8x8) you have to place eight queens without threatening each other. In the game of chess, the queen threatens those pieces that are in the same row, column, or diagonal. The most obvious way to solve the problem is to systematically test all the ways to place eight queens on a board using the coordinates (x, y); for each queen it would be implemented with a double loop For, and only in creating those environments (without checking the solutions) we would have a total of 64^8 possibilities. More than 281 billion (millions of millions) of cases. Following without even checking if there are threats between queens, among all those cases it would be possible to place different queens in the same box, which obviously is not correct. The first improvement would be to avoid repetitions and take care of the placement order from the upper left to the lower right; thus, the number of situations to be examined

$$\binom{64}{8} = 4.426.165.368$$

but they are still many and the requirements are not yet verified. A first improvement of this last method explained would be to never put more than one queen in a row. This reduces the representation of the board to a vector of eight elements, each of which gives the position of the queen within the corresponding row (queen K will be in row K and column v[K]). With this situation you would have 8 possibilities for each queen, that is 8^8=16,777,216. We have not only reduced cases, it is also that we have managed to move towards the solution by avoiding a threat situation. Now we will avoid that two queens are in the same column v[K]. Once we realize that with the representation of the board through a vector prevents us from trying to put two queens in the same row, it is natural to realize that we can also do it with the columns, therefore, we will now represent the board by means of a vector formed by eight different numbers between 1 and 8, by means of a permutation of the first eight integers:

- for the first there are 8 possible positions (columns),
- for the second, there are only 7 positions, etc.

The number of cases is 8! = 40,320 possibilities. It is at this time that we can start thinking about Backtracking. Instead of creating those 40,320 possibilities and testing, we are putting one queen each time: if there are threats we do not follow, and if there are no threats we continue with the recursion. Thus, we avoid generating cases like those that begin with (1, 2, x, x, x, x, x, x) because queen 1 and queen 2 share the same main diagonal and threaten each other; it does not make sense to continue. We will use the following structures:

- A col set that represents the columns occupied by the queens used so far, with values between 1 and 8. It has the same values as the solution to the problem.
- A diag45 set that represents the 45º diagonals of the board. A diagonal of 45º has the property that the sum of its coordinates is constant, for example, the squares (2, 5) and (4, 3) are in the same 45º diagonal since the two coordinates add 7. The set diag45 Take the values between 2 and 16.
- A set diag135 that represents the 135º diagonals of the board. These diagonals have the property that the difference of the first coordinate minus the second

coordinate is constant; the squares (2, 4) and (5, 7) are on the same 135º diagonal. The values of the set are between -7 and 7.

The pseudocode would be:

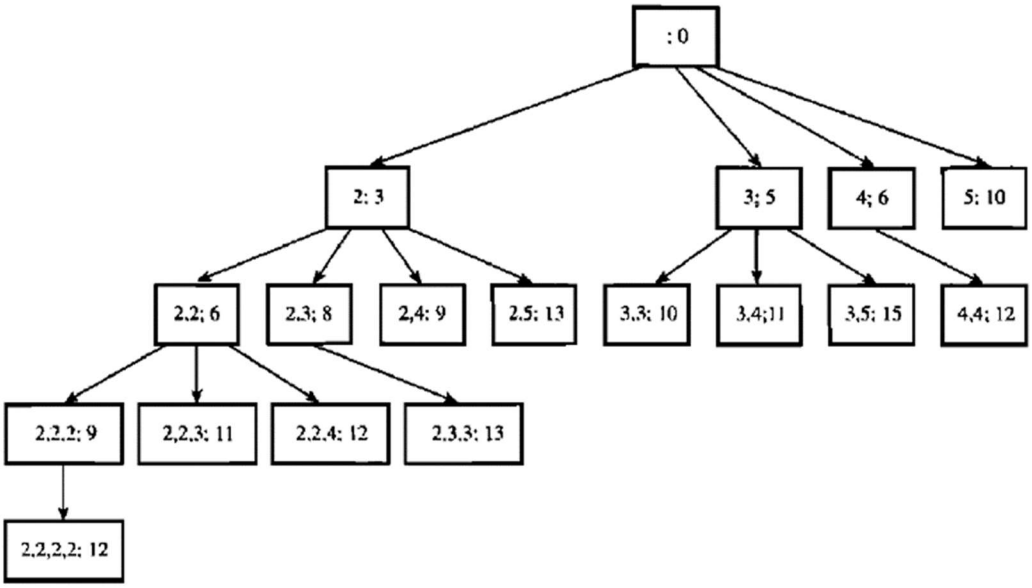
```
proc queens(I k: integer, I col, diag45, diag135: set, I/S sol: vector)
    if (k > 8)
        Print(sol)
    else
        for j = 1 to 8 do
            if (j∉col) and (j+k∉diag45) and (j-k∉diag135)
                sol[k]= j
                queens(k+1, colU{j}, diag45U{j+k}, diag135U{j-k},sol)
            eif
        efor
    eif
eproc
```

and the initial call is: queens(1, Ø, Ø, Ø, sol_vacía).

By the way of passing the environment structures it is not necessary to undo actions, and the local copy is made in the execution of the recursion. The previous procedure needs 2,057 nodes, and it is enough to explore 114 to find a first solution.

5. The problema of the backpack.

We have n types of objects and an adequate number of objects of each type. For i=1,2,...,n an object of type i has a weight w_i and a positive value v_i. The backpack can carry a weight that does not exceed W. Our goal is to fill the backpack maximizing the value of the objects included, without exceeding its capacity. We can take an object or discard with it, but we cannot take a certain part of an object. Let us suppose 4 types of objects of weights 2, 3, 4 and 5, whose values are respectively 3, 5, 6 and 10. The value of W is 8. Exploring the implicit tree:



The objects are loaded in the backpack in order of increasing weight. The figures to the left of the semicolon are the weights of the objects that we have decided to include. On the right is the current value of the load. Going down from one node to one of your children corresponds to deciding what kind of object is going to be put in the backpack next. Any node, such as (2,3; 8) corresponds to a partial solution of our problem. Initially, the partial solution is empty. The backtracking algorithm explores the tree as a journey in depth, building nodes and partial solutions as it progresses. In the example, the first node visited is (2; 3), the next is (2,2; 6), the third is (2,2,2; 9) and the fourth (2,2,2; , 2; 12). As each new node is visited, the partial solution is extended. After visiting these 4 nodes, the route is blocked: the node (2,2,2,2; 12) has no unvisited successors, since adding more elements to this partial solution would violate the capacity restriction. In addition, as it could be the optimal solution of our case, we memorize it. The algorithm goes back now looking for other solutions. In each step retraced by the tree, the corresponding element of the partial solution is eliminated. Go back first to (2,2,2; 9) which also lacks successors. Go back to (2,2; 6), which has two successors: (2,2,3; 11) and (2,2,4; 12) that do not improve the previously memorized solution. Go back another step, and so on. In (2,3,3; 13) we find a better solution than what we already have, and then (3,5; 15) is even better, and will end up being the optimal solution of the case. The algorithm is the following, assuming that the data n and W, and the vectors w [1..n] and v [1..n] are global variables:

```
function backpack(i,r)
    {Calculates the value of the best load qith objects of type i to n and whose
    total weight is lower than r}
    b=0
    {All the posible objects are tried by turn}
    for k=i to n do
        if w[k]≤r then
            b=max(b,v[k]+mochilava(k,r-w[k]))
    return b
```

The initial execution is: backpack(1, W). Each recursive call corresponds to extending the depth path to the immediately lower level of the tree, while the loop to examine all the possibilities of a given level. The composition of the load that is being examined is implicitly given by the values of k in the recursive stack.

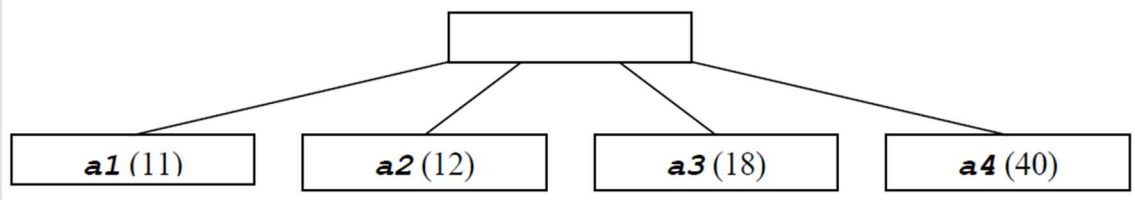
6. Branch and bound.

As the backtracking, branching and bounding is a technique to explore an implicit directed graph, which is usually acyclic (or even a tree). It is about looking for the optimal solution of a problem among all the possible solutions. In each node, we calculate a bound of the possible value of those solutions that could be found later in the graph. If that level shows that any of these solutions is worse than the one best found so far, we will not continue to explore that part of the graph. Sometimes, the calculation of dimensions is combined with a path in width or depth, and serves to prune certain branches of a tree (or close certain paths of a graph). Other times the calculated elevation is also used to select the path that, among the open ones, seems more promising to explore first. Deep searches explore the nodes in reverse order of their creation, using a stack to store the nodes that have been generated but have not yet been examined. The amplitude searches finish exploring the nodes in the order in which they are created, using a queue to save the nodes created and not yet examined. The branch and bounding method uses auxiliary calculations to decide at each moment which node should be explored and a priority list (ideally implemented with a heap) to store the generated and not yet examined.

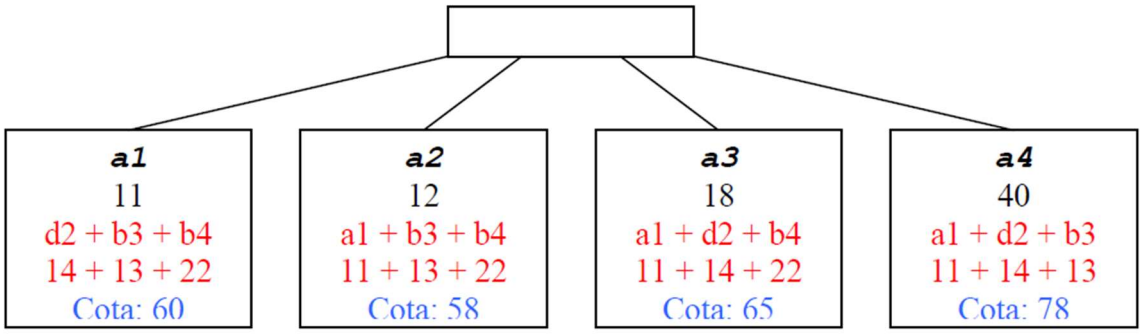
An example of branching and bounding is the problem of allocation: this problem consists in assigning N tasks to N agents, so that each task is done only once and each agent has a single task associated with it. Each assignment has a cost M[i, j], where i is the agent and j the task, so of the N! possible valid assignments, it is necessary to determine the one that has a total cost as low as possible. Suppose that the cost matrix is the following:

	1	**2**	**3**	**4**
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

We start by creating the boundaries: we are going to obtain a real level, based on known solutions that allow us to bound undesirable candidates, and another estimate that serves to quickly end the problem if there are candidates with values close to it. For this example, we will use as a real value any valid assignment that is easy to find; for example the solution a1, b2, c3, d4, which has cost 11+15+19+28=73. Any partial candidate that has a cost higher than 73 will be rejected (bounded) since instead of continuing to generate it we could stay with the complete solution already found. As an estimated value we will consider, for example, the minimum cost that would be on the part of the agents. The minimums of the rows would be 11(a1)+13(b3)+11(c1)+14(d2). It should be noted that this is only an estimate of the best that could happen, but that it is not a real solution (task 1 would be assigned twice), and would cost 49. But we could also have considered the minimum cost to perform the tasks, that is, the minimums of columns 11(a1)+12(a2)+13(b3)+22(b4)=58. Considering both estimates, the estimated value will be max{49, 58}=58. Thus, the range of valid values to explore will be [58, 73]. To carry out the branching and bounding we will go through an implicit tree in which the nodes correspond to partial assignments of agents to tasks. In the root of the tree there is no assignment. At each stage (tree level) one more agent is assigned. The possibilities for the first assignment would be:

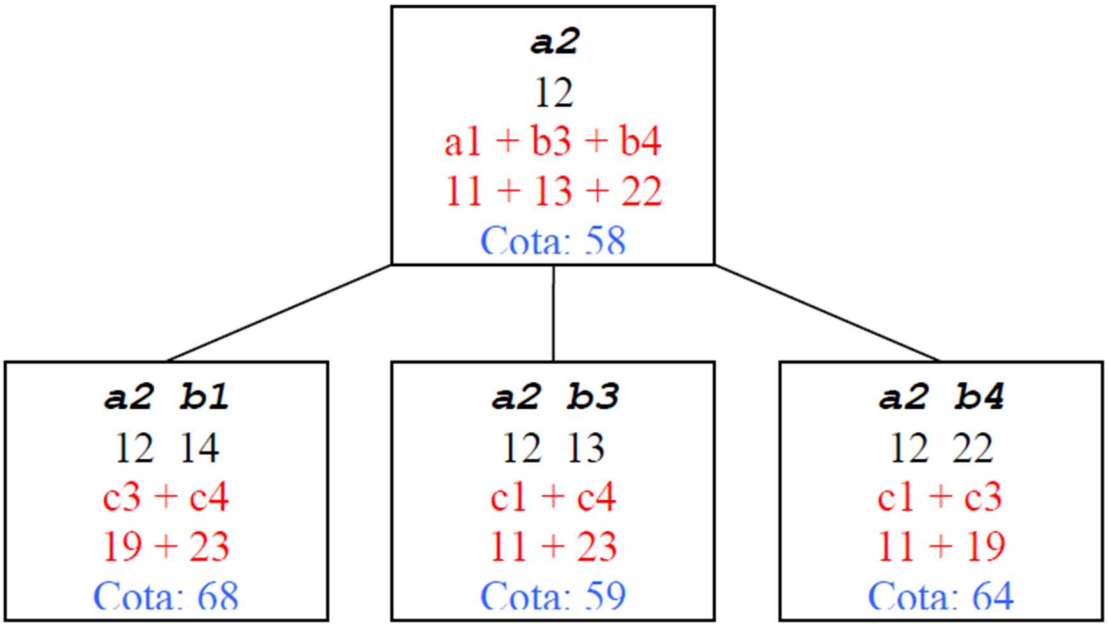


In order to decide which of the candidates the tree scan continues with, each node calculates a level of the best solution that could be obtained from that node. We will consider the minimum of the necessary costs to carry out the tasks that still need to be assigned; for example, after assigning a1 we estimate the best cost of doing task 2 (which would be d2, cost 14), task 3 (b3, cost 13) and task 4 (b4, cost 22). Again, this is an estimate and does not form a complete possible solution. We add this information to the nodes (in black, the decisions taken, in red, the estimates, in blue, the level obtained):

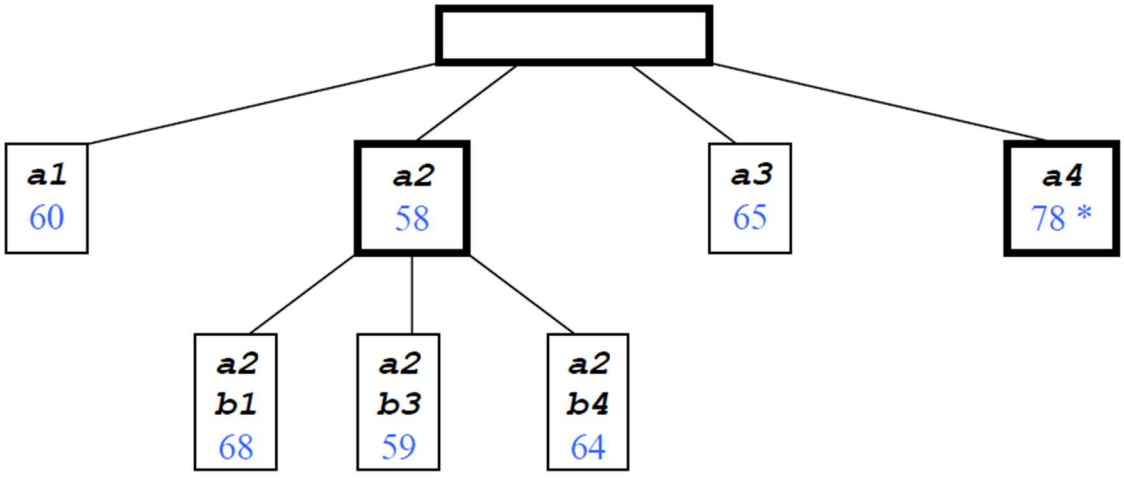


Of these four nodes, we can see that the last one (the one that starts assigning a4 and that is not even a complete solution) in the best case has a cost of 78, which is higher than the known upper bound (and which is a valid solution). Therefore, that node does not need to be scanned and bounded. Of the other three nodes, called living nodes, the one with

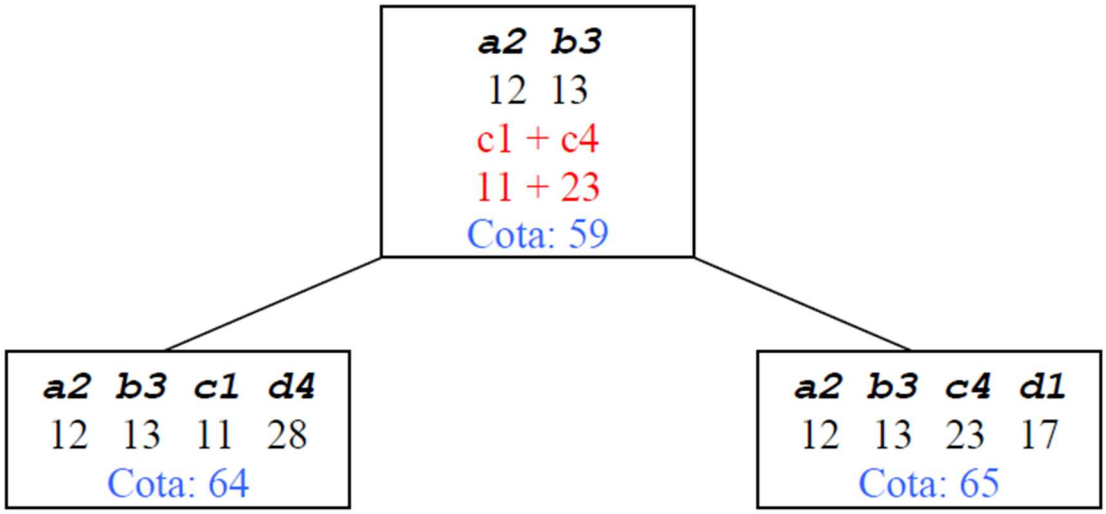
the best available elevation is chosen and the exploration is continued, trying to assign the agent b.



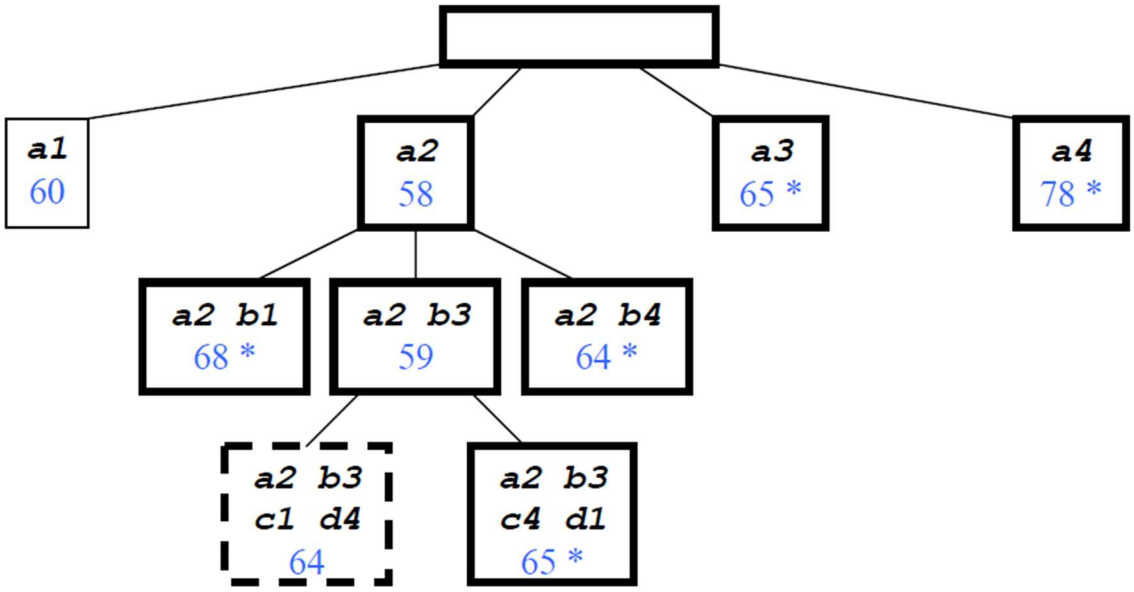
Now the simplified structure of the tree (remember that only the nodes that have not yet been explored are saved) would be as follows, with the bounded nodes marked with * and those no longer in the priority queue with dark border.



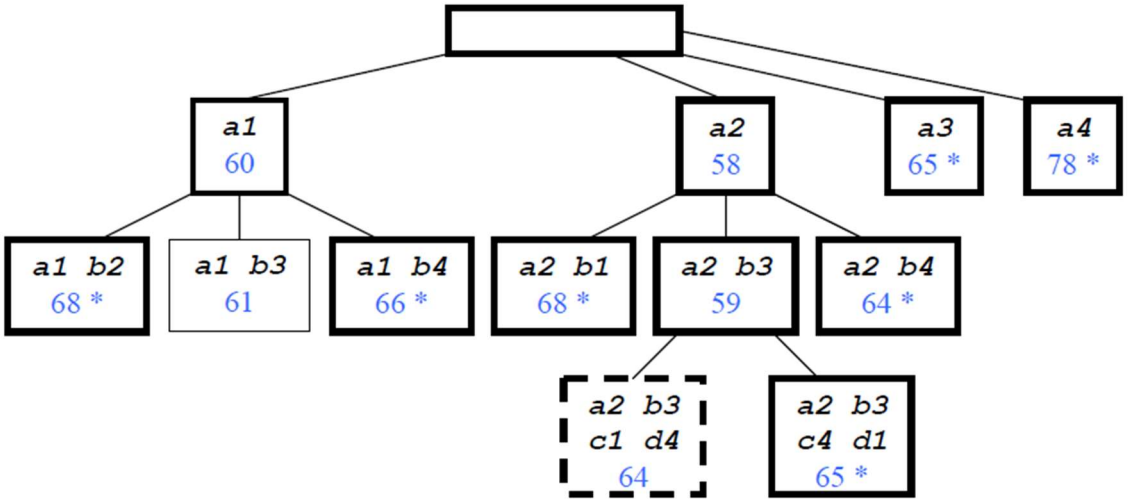
Of the live nodes, the one with the smallest dimension is taken and another agent is added, and since there would only be one task left to perform the estimation, it is really a complete solution.



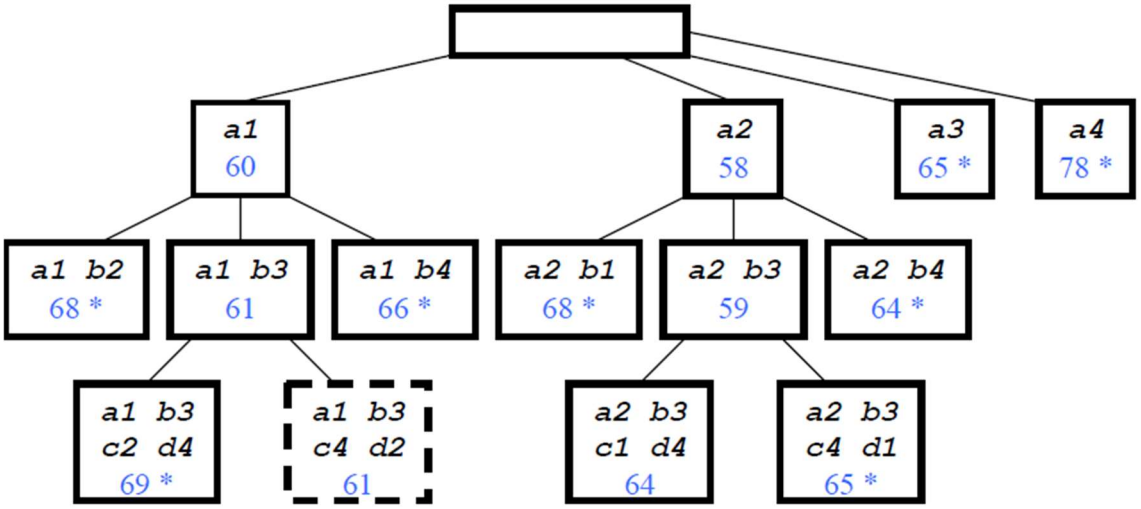
Having found a better solution than the known one, the upper bound is updated to 64, so the range of admissible values is [58, 64]; this causes that there are more nodes that are bounded automatically, and the tree would remain (dotted the optimal solution until now):



It can be seen that having found this solution a2 b3 c1 d4 the tree has been very reduced at the time of exploring live nodes, leaving only one. We finish the example with the last two implicit trees developed after a1:



and after developing a1 b3 is what will finally be the optimal solution



If a complete search had been made it would have been necessary to go through a number of nodes equal to 4+4*3+4*3*2=40 nodes, while with the branch and bounding only 14 nodes had to be created.

The need to maintain a list of nodes that have been generated but have not been explored in their entirety, located at different levels of the tree and preferably arranged in order of the corresponding boundaries, makes the branching and bounding difficult to program. The heap is an ideal data structure to store this list. Unlike the depth path, and related techniques, the programmer does not have an elegant recursive formulation of branching and bounding. However, it is usually used in practice.

When we use this technique we must reach a balance in the resources dedicated to the calculation of the elevation, although it is usually profitable to invest a reasonable time in its calculation. Although it is not possible to assure how well this technique will behave in a given problem, it is more likely that a better level will cause us to examine fewer nodes and reach the faster solution. Its drawback is that we will have to spend more time in each node calculating the corresponding dimension. In the worst case, it could happen

that an excellent boundary does not allow us to cut any branch, wasting the work of its calculation in each node.